23｜Raft：分布式系统间如何达成共识？

2022-02-12 黄清昊

《算法实战高手课》 课程介绍 >

讲述：代录（老师近期生病）

时长 17:03 大小 15.62M

你好，我是微扰君。

今天我们要来谈一谈分布式系统中一个非常重要的问题：分布式共识问题，也就是一致性问题。

我们知道，分布式系统的诞生，主要是为了提供单机无法进行的计算和存储、提高吞吐量、增加容错性等。而在现在的互联网架构下，分布式系统由于大量使用廉价的商用机器，节点故障是不可避免的。

在这种情况下，多个机器如何像一个整体一样工作，是件很困难的事情，这里一致性算法就起到了至关重要的作用。

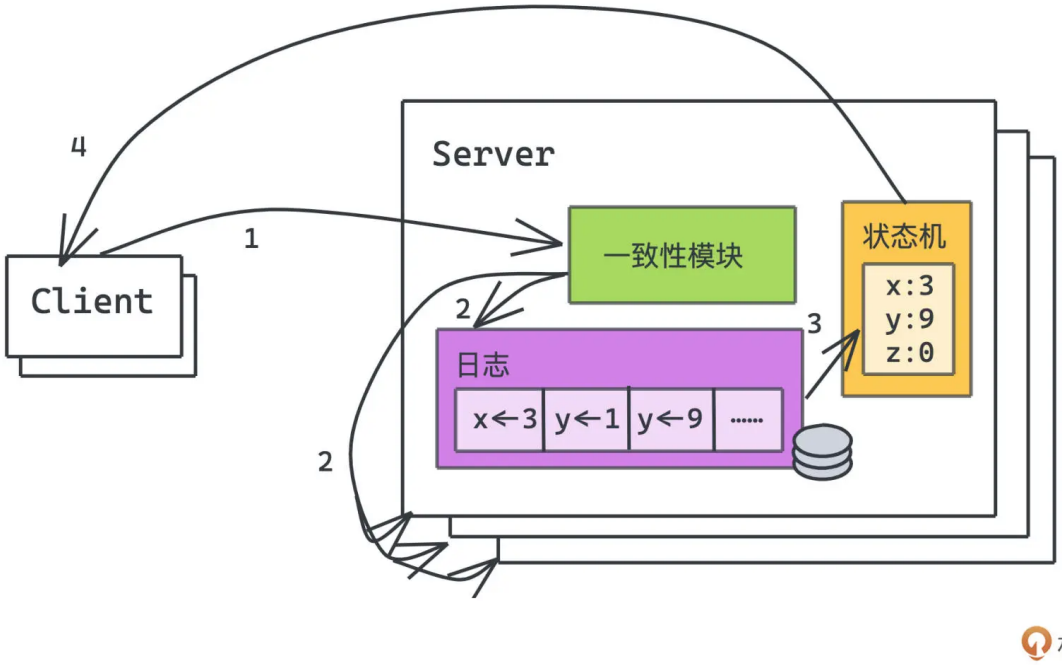
领资料

以分布式 KV 存储系统为例，我们来先搞清楚分布式一致性到底解决的是什么问题。

复制状态机

之前在操作系统章节中讲过，数据库常用 redo-log 来实现事务等能力，那当这样的存储系统不再是单机节点，我们通常也需要采用多台机器来存储日志，把同样的日志在不同的节点都存储一份。这样如果有一台节点挂了，整个系统还可以用备份节点来提供日志的能力。

让多台服务器存储相同顺序的多条相同指令，也就是“日志”，可以帮助我们实现复制状态机。



每一个状态的变更记录都会先在日志中存储并 commit，之后才 apply 到状态机中修改对应的状态，这个设计能在分布式系统中解决许多和容错性相关的问题。

既然涉及多个节点存储同样的一份东西，怎样才能保证多个独立的节点所存储的内容是一致的呢？这就是我们常说的“一致性问题”了。

日志一致性问题

客户端通常只会向分布式系统中的某个服务器发起请求，然后由这个服务器的一致性模块，在多个复制状态机之间进行消息的同步，正常情况下，多个节点同步都会成功，这样不同节点的日志自然也都是一致的，所有的节点都会以相同的顺序包含相同的请求，从外界看起来，行为也就像是一台机器一样。

但是在服务器发生故障时，依然要保持正确的复制变成了困难。

一种最暴力的做法可能是：定义一个主节点，每一次写请求都请求到主节点，等主节点一致性模块向集群中的每个节点都成功写入了同样一份数据，再返回给客户端成功的消息，进行消息的 commit。只要有一个失败，就不进行消息的 commit。

但是我们看这个方案，显然不是很高效。无论是存在慢节点，会导致整体响应被拖的很慢，还是一台节点挂了，会导致整个集群不可用，都是不可接受的。

那我们分析一下在实际系统中，一致性算法通常会存在哪些问题：

1. 在分区、网络延迟、乱序等情况下都需要保证安全性，也就是说需要数据一旦返回，一定是正确的。
2. 可用性问题。部分节点故障，但在集群中大部分节点可运行且能互相通信的情况下，要保证整个系统是可以工作的。
3. 不依赖时钟保证一致性。因为物理时钟在分布式系统中是不可信的，不同节点间的时间并不同步的，而且随时可能因为时钟同步而导致时钟回跳等等。
4. 慢节点，要求不能影响系统整体性能。

接下来我们要学习的 Raft 算法就很好地考虑了这些问题。

Raft 与 Paxos

Raft 提出之前，在非拜占庭条件下，分布式一致性领域里，Paxos 算法，一直占据着统治性的地位，但 Paxos 是出了名的难理解，工程实践也比较困难。

拜占庭条件源自一篇论文，也是 Paxos 的作者写的，他用一组将军围困一座城市的假想问题，描述当点对点通信的节点中出现了恶意节点传播不正确消息时，达成共识的困难处

境。非拜占庭条件指的就是所有节点都是正常工作，只可能出现消息不可达，不会有消息错误的情况。拜占庭将军问题本身也非常有趣，你感兴趣的话可以自行搜索了解一下。

Paxos 的艰深难懂其实也是 Raft 算法提出的主要动机。Raft 和 Paxos 都是只要有超过一半的服务器可以运行并互相可通信，就可以保证整个系统可用。

和 Paxos 不同的是，一致性问题，**被 Raft 明确拆解成了三个比较独立的、更好理解的子问题**，并且团队在许多实现细节上做了很多努力和权衡，也增强了系统里的许多限制，简化了需要考虑的状态，尽量让过程和接口的设计变得清晰易懂。比如，Raft 中就引入了 Leader，由 Leader 进行全局消息的把控，也不允许日志中存在空洞的情况，都是一些比较常见的权衡。

接下来我们就来逐一学习 Raft 拆解出来用于达成分布式共识的三个子问题：领导人选举、日志复制、安全性。

子问题一：领导人选举

Raft 引入 Leader 的概念，也就是领导人节点的思路，和两阶段提交里的协调者性质其实差不多的。

本质上，都是因为在分布式的系统中，各个节点不具备全局的信息，那为了感知到不同节点对请求的响应情况，我们通常就会引入一个主节点，由它进行统一的控制和调度，这样整个分布式的处理逻辑就会变得比较简单。

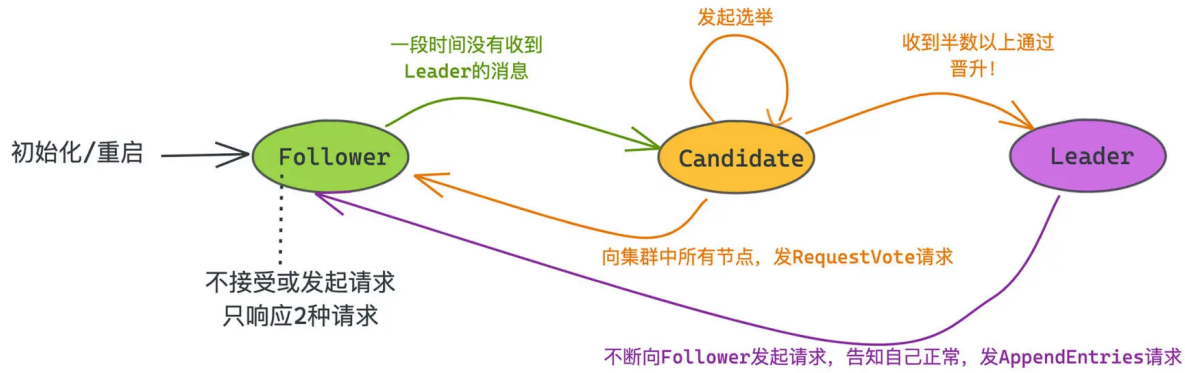
在 Raft 中，Leader 就负责接收客户端的请求，由它统一向其他节点同步消息，等收到半数的节点 Commmit 日志的响应后，就会把状态应用到状态机，并返回。

思路很简单，但是 Leader 节点如何被选出呢？

Raft 设计了一套节点状态机制，每个节点永远处于三个状态之一：

> Follower 追随者：所有节点初始化或重启的时候都处于 Follower 状态。它不会接受请求，也不会发起请求，只响应由 Leader 发起的 AppendEntries 和 Candidate 发起的 RequestVote 请求。

Candidate 候选人：是 Follower 晋升为 Leader 的中间状态，从语义上就能看出来这个阶段是需要投票的。Follower 如果在一段时间没有收到领导人的消息，就会变成 Candidate 并发起选举，也就是向集群中所有节点发出 RequestVote 请求，如果收到半数以上也就是 (n/2+1) 的通过，就可以成功晋升为新的 Leader 。

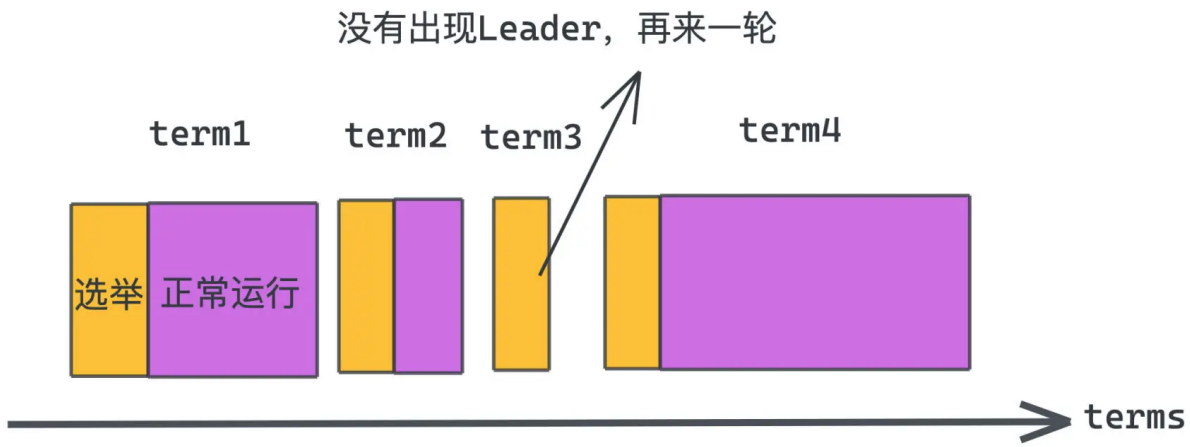


极客时间

Leader 领导人： 系统大部分时候只有一个节点处于 Leader 状态，如果有两个节点同时处于 Leader 状态，也最多只有一个是真正有效的。Leader 会不断的向 Follower 发起请求，告知它们自己还在正常工作。这里的请求就是后面用于复制日志的 AppendEntries 请求，我们马上展开讲解。

每一任新的领导人出现，都会带有一个任期（term）。任期时长不确定，只要网络不发生大面积分区，而且超过半数的节点和 Leader 一直可以正常工作，这届任期可能就会非常长。

任期编号是单调增的，1、2、3……，在每一次选举发起的时候产生。



候选人发起选举的时候会把当前的任期加 1，再发给其他节点投票，如果其他 Follower 收到的请求带的是过期的任期，就会直接拒绝这次请求，对应的 Leader 和 Follower 发现后也会立刻变成 Follower 状态，因为这意味着此时已经出现过一个任期更高的合法 Leader 了。

大致设计就是这样，但是我们把这套规则应用到真实系统中就会存在一些问题。你可以先暂停思考一下预计会出现哪些问题，如何解决，我们再一起讨论。

首先当同时有多个 Candidate 产生，发起票选，**每个 Candidate 的票数都不足 n/2+1 的时候会发生什么呢**？

这个很简单，平局收场，Term 再加 1，我们重新选举一轮就可以。所以 Candidate 也会维护一个定时器，用于处理这种超时的情况。网络分区中的 Candidate 可能会不断的提高自己的 Term，但是因为它没有任何新的数据被写入，网络恢复的时候它自己也能很快感知到这点，从而恢复到 Follower 的身份。

看到这里你可能又想到新问题了：如果碰到了这样的情况，**多个 Candidate 一起超时，又会触发下一轮票选瓜分，岂不是永远选不出 Leader 了**？

解决这个问题的方法，也很简单常用。我们可以在选举超时时间中引入一定的随机性，而不是一个固定值，比如 150-300ms，这样多个 Candidate 在下一轮超时的时候，肯定就会错开发起选举请求的时间了。

当然，这也需要我们保证有良好的网络环境，选取发出 - 被收到的时间一定要比较短才行。

```
1 广播时间（broadcastTime） << 选举超时时间（electionTimeout）
```

复制代码

不过这里还有一个问题不知道你有没有思考，**为什么我们一定要要求半数以上的票选才能晋升呢**？

这其实也是在分布式系统中非常常见的做法。容斥原理我们知道，能获得半数以上票选的候选人只可能有一个。所以半数机制，如果遇到网络分区，网络分区少数的那一方就肯定不会产生 Leader；同样，同一个 Term 下全局也只会有一个 Leader，不会出现脑裂也就是同时有多个 leader 的情况。

日志复制

现在 Leader 选举完成，它就要扛起接受客户端请求并复制日志的大旗了，主要职责就是发起 AppendEntries 请求，这里的 Entries 主要指的就是日志记录，它可以做两件事。

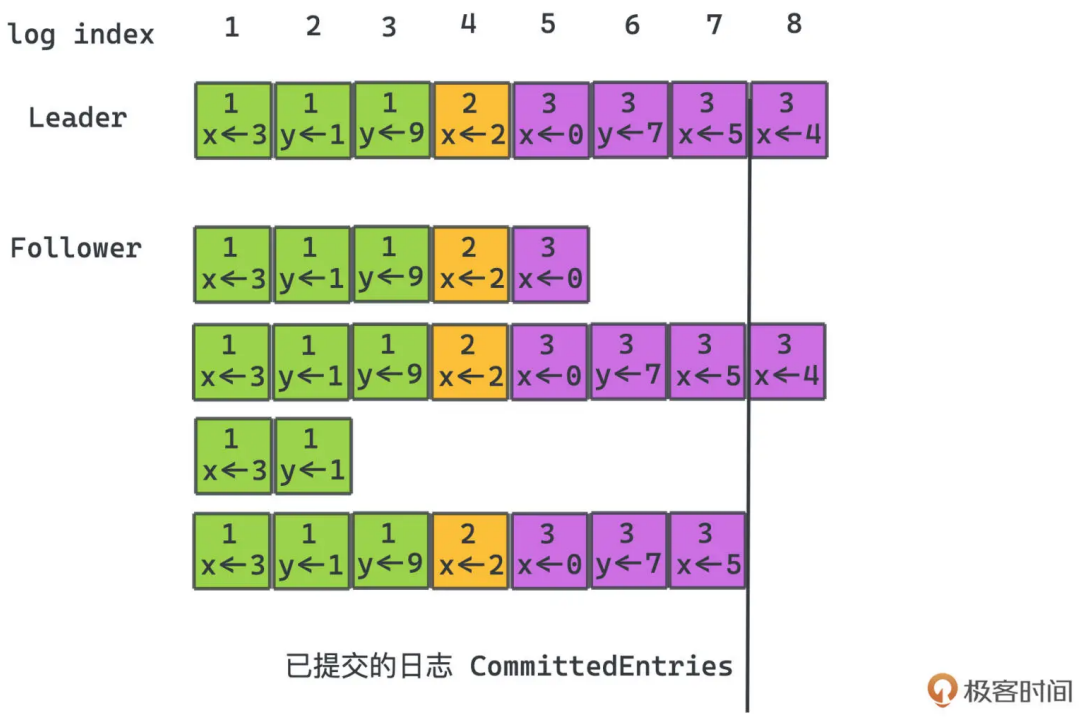
第一就是我们前面说的，Leader 需要周期地告知其他 Follower 节点，自己还在正常工作。Raft 的实现，就是让 Leader 不断地向其他节点发起空的 AppendEntries 请求。

当 Follower 收到这样的请求时，只要请求的任期没有过期，Follower 就会接受这个请求，知道 Leader 还正常工作，自己也就没有必要揭竿而起，成为 Candidate 了。这个和很多系统中的心跳机制是一样的。

第二，也是 AppendEntries 的主要用途——复制日志，和字面意思一样，就是由 Leader 发起，要求 Follower 在日志中追加记录的意思。

当 Leader 接收了客户端的请求，它就会并行地请求其他节点，带上客户端请求的指令，要求其他节点进行复制并返回结果。**当 Leader 觉得日志被安全地复制了之后，才会将指令应用到状态机中并返回客户端。**

什么是安全的复制呢？就是指一旦决定这个日志可以被应用到状态机（我们也叫“已提交的日志 CommittedEntries”），即使之后任何节点出现不可用的情况，已提交的日志一定不会丢失，且最终会被集群中所有正常工作的节点应用到状态机。



而 Raft 对“已提交”的条件定义也很简单有效，**如果一个日志被 Leader 复制到大多数节点，日志就算被提交了**，反之则没有，还有可能被其他更新的任期的日志所覆盖。这一点约束我们稍后马上会展开讨论。

另外，Raft 在记录日志的时候，除了会记录日志任期、具体操作，也会给每条记录都赋予一个日志索引，这样可以帮助节点定位自己所持有的日志具体是哪些。

Log Matching

有了 index 和 term 的概念，Raft 通过引入一些约束，使得所有的日志始终拥有着“日志匹配”的特性，主要是两条规则：

1. 不同日志中的两个记录，如果拥有相同的任期和索引，它们的内容相同。
2. 不同日志中的两个记录，如果拥有相同的任期和索引，它们之前的内容也相同。

前面我们已经说了，Raft 同一个任期里肯定只有一个 Leader，如果这个 Leader 写了某条日志，它在不同节点上日志的索引一定是相同的。

这是因为 **AppendEntries 被接收时，会执行一致性检查**，Leader 提交请求的时候会带上自己的 prevLogIndex 和 prevLogTerm，表示上一个日志条目的索引和任期。如果 Follower 发现自己的日志里找不到这个任期和索引对应的条目，会拒绝此次 AppendEntries 请求，这个就是 Raft 协议很关键的一个约束；所以当 AppendEntrires 成功时，Leader 能保证 prevLogIndex 之前所有的记录都是相同的。

这个逻辑你仔细顺一下就很清晰了，具体的证明有点像数学归纳法，初始状态是满足日志匹配的，如果执行了一致性检查，那么后续的所有状态，日志匹配特性也都是满足的。

如果 Leader 和 Follower 由于崩溃，出现日志记录不同的时候，Leader 就会要求 Follower，按照自己的日志覆写。Leader 为每一个 Follower 都记录了一个 nextIndex 的字段，表示下次应该发给 Follower 的日志，在 Leader 刚刚晋升的时候，Leader 就会将这个值初始化为自己的最后一条日志的索引 +1。

如果 Follower 日志和 Leader 日志有所冲突，Leader 会尝试减小 nextIndex 的值，直至两者 nextIndex 所对应的日志相同；此时，LeaderAppend 的记录就包含了从 nextIndex 开始的全部日志，Follower 收到之后就会把不同的部分覆写；如果成功，Leader 也会修正 nextIndex 的值。

基于这样的约束，**整个覆写的过程一定是单向的，只会发生在 Follower 节点上**，Leader 从来不会修改自己的日志。

安全性

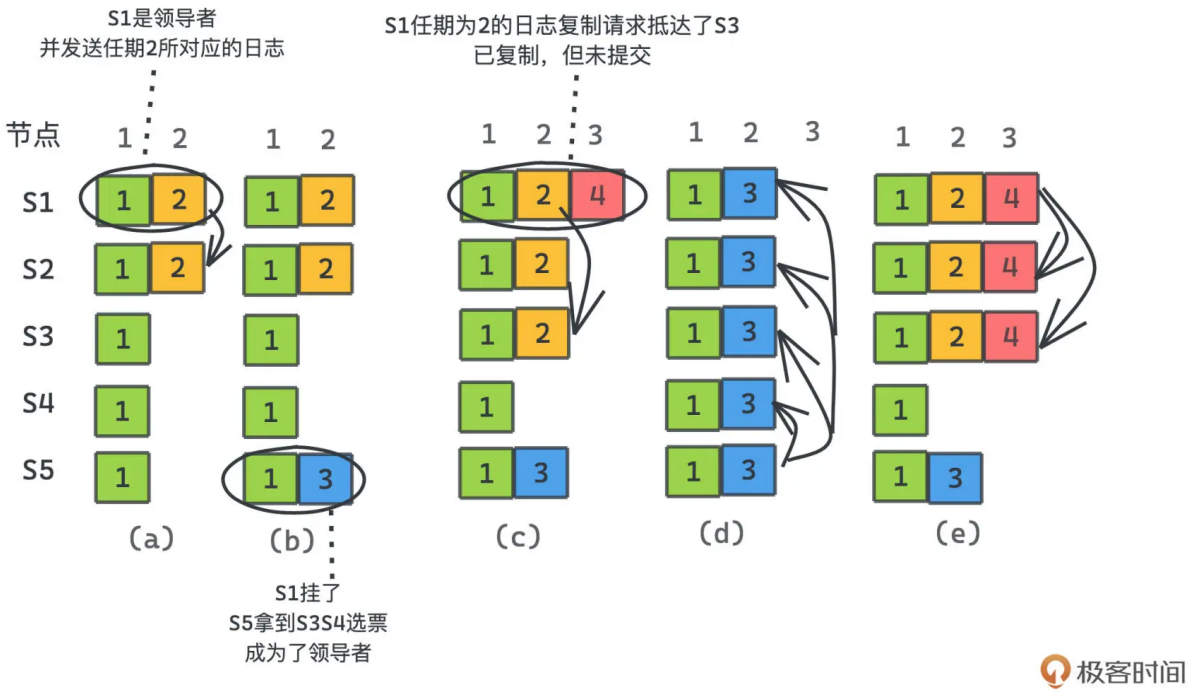
但到目前为止，Raft 协议还有一个重要的特性没有得到保证，就是“领导人完整性”，也就是需要保证：如果某个日志条目在某个任期号中已经被提交，该记录必定出现在更大任期号的所有领导人中。

这是一个非常重要的特性，不然会无法保证某个领导人在任期内提交的日志不会被后来者所覆盖。Raft 对这个问题作出了另一个简单的限制，相比于一些其他的一致性算法，显得更加清晰。**限制 Candidate 提交选举请求的时候，必须至少和 Follower 的日志一样新，才可以获得选票。**

这就意味着，如果 Candidate 获得了超过半数的选票，说明至少有半数的 Follower 节点，日志条目和自己一样新，而所有 commit 了的记录，也一定在半数的节点中出现了。

根据容斥原理，半数同意的节点一定会和 commit 了某个日志的节点有所重叠，新的 Leader 至少拥有了所有被 commit 日志一样新的日志。这样，被 commit 的日志，一定会被更大任期的领导所包含。

我们结合论文中一个比较清晰的例子来理解。一开始，S1 成为了任期为 2 的领导者，并开始发送任期 2 所对应的日志，随后在 b 时刻，S1 挂了，此时，S5 拿到了 S3、S4 的选票成为了领导者。



这个时候 c 时刻，S1 任期为 2 的日志复制请求抵达了 S3，并进行了复制，但还没有提交：

如果 S1 在 d 中又崩溃了，S5 就可以像 d 那样覆写所有之前任期的日志，d 时刻；

如果在崩溃之前，S1 就对多数节点日志进行了复制并提交，则 e 就不再有被选举上的可能性，因为多数节点都拥有更新的日志，e 时刻。

原论文中还提到了一些简单的优化，比如日志压缩、采用 Chubby 和 Zookeeper 用的快照技术等，减少因为日志增长越来越多空间被占用和对应的同步成本问题等等，如果你感兴趣可以看看原论文。

总结

Raft 协议，除了各种协议细节，今天学习的几个比较有价值的技巧对你工作也很有帮助。

我们为了避免票选被多个同时称为 cadidate 的节点平分，进入无限循环，可以在选举超时时间里引入随机性，避免多个节点继续在同一时间发起票选请求。分布式系统多个节点存在时，往往会采用奇数的节点，这样就可以通过少数服从多数的机制，在集群中保证同一时间只会有一个主节点了。

Raft 将复杂问题拆解成多个明确清晰的子问题，分而治之，也是一种系统设计的哲学，你可以好好体会。

Raft 算法逻辑还是比较清晰的，但是有很多细节和边界问题需要我们反复琢磨，不自己动手实践一遍，理解程度一定还是比较有限，所以这里也推荐 MIT 6.824，供你练手学习，Lab 就是基于 Raft 和 Golang 语言实现一个简单的分布式 KV 存储（18 年我做过一次，当时有几个 case 没有跑过，就弃坑了，但整体还是收获很大，而且过程颇为有趣，祝你也能研究顺利）。

课后作业

最后也留一个思考题给你。Raft 在现实的工程实践中还有许许多多的优化，不知道你听完了今天的讲解之后有什么觉得可以优化的想法吗？

这里给你抛砖引玉一下，Leader 给 Follower 同步日志的时候，nextIndex 是一步步向下尝试的，如果中间缺失了很多日志，效率其实可能比较低？你有什么办法吗？

如果你还有想法也欢迎在留言区和我讨论。如果觉得有帮助的话，也请转发给你的朋友一起学习。我们下节课见。

分享给需要的人，Ta购买本课程，你将得 20 元

生成海报并分享

赞 1 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 22 | PageRank：谷歌是如何计算网页排名的

精选留言

写留言

> 由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。